Provably Correct Runtime Monitoring

Irem Aktug, Mads Dam, Dilian Gurov

CSC KTH
Stockholm, Sweden

KTH Computer Science and Communication

Outline

Mobile Code

Applications obtained from an untrusted source to be executed locally

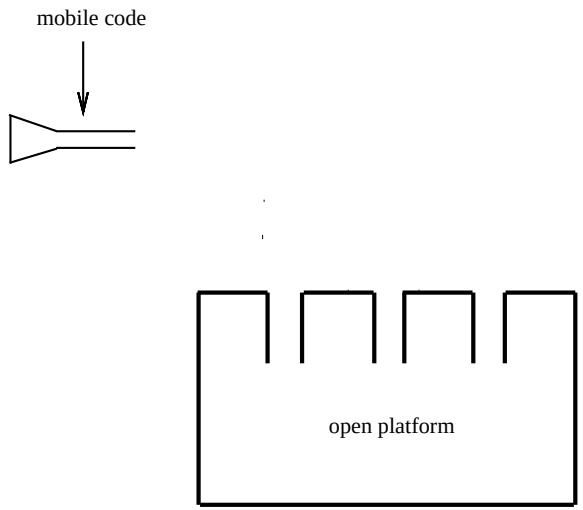


Figure: Mobile Code Joining Platform

Examples:

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Mobile Code

Applications obtained from an untrusted source to be executed locally

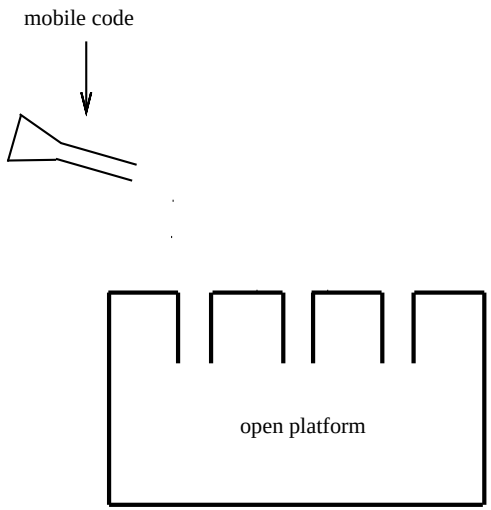


Figure: Mobile Code Joining Platform

Examples:

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Mobile Code

Applications obtained from an untrusted source to be executed locally

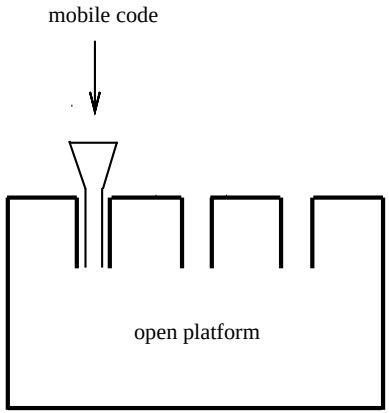


Figure: Mobile Code Joining Platform

Examples:

- Java Web Applets
- Code embedded in Microsoft Office documents
- Java Card Applets (smart card applications)
- Java Midlets (mobile phone applications)

Mobile Security

Main Problem: How do we know if the system combined with the mobile component works "correctly"?

Mobile Security

Main Problem: How do we know if the system combined with the mobile component works "correctly"?

Common solution

Identify property P to be satisfied by mobile component.
Examples:

- *Sends SMSs only with user permission*
- *Does not connect to internet if personal data is accessed*

Mobile Security

Main Problem: How do we know if the system combined with the mobile component works "correctly"?

Common solution: Identify property P to be satisfied by mobile component.

Subproblem: How do we check that the mobile component has property P at time of join?

Mobile Security

Main Problem: How do we know if the system combined with the mobile component works "correctly"?

Common solution: Identify property P to be satisfied by mobile component.

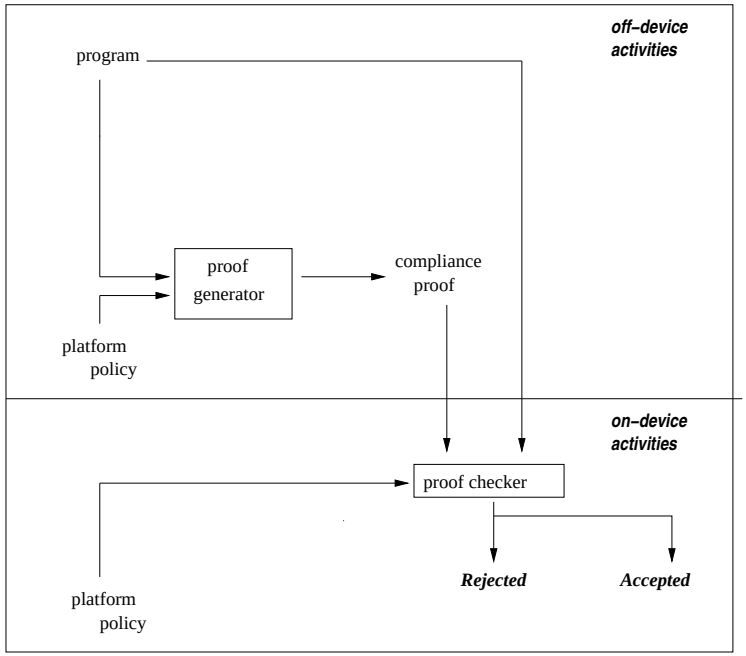
Subproblem: How do we check that the mobile component has property P at time of join?

Common solutions

- Code signing
- Sandboxing
- Proof-carrying code

Proof-carrying Code

Necula and Lee'96: Proof checking is easier than proof construction yet as sound.



The S3MS Project

(Security of Software and Services for Mobile Systems) http://www.s3ms.org

- European Union project that aims to develop novel approaches for security of mobile devices such as mobile phones, PDAs.
- 15 partners, both from industry and academics.
- The main objective is to safely run third-party applications on mobile platforms.

The Target Applications

We consider java bytecode programs.

As a first step, we handle:

- only sequential programs,
- exception handling, and
- inheritance.

Example: Target Program

Label	Instruction
L1	`aload r0`
L2	`getfield gui`
L3	`invokevirtual GUI/AskConnect()Z`
L4	`ireturn`

Figure: An application method

Policies

a.k.a "How to define safe?"

A *policy* is a predicate on the set of all possible sequences of actions and selects only the acceptable sequences.

Our security relevant actions are calls to and returns from a fixed API.

"*Don't send after read*"

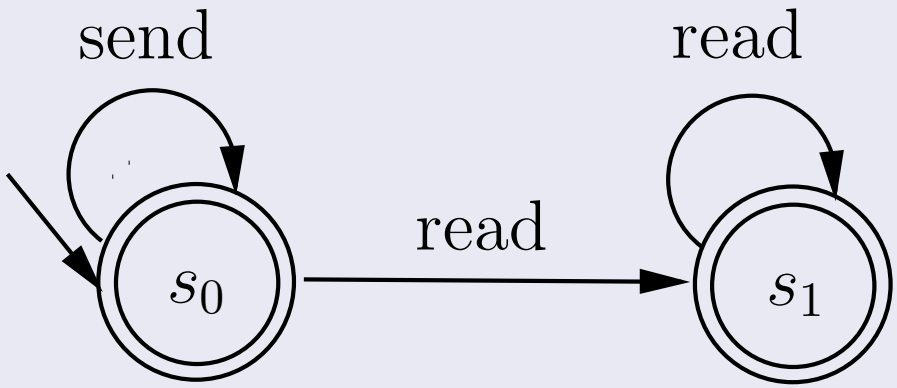


Figure: Security Automata for The Policy

"Don't send after read"

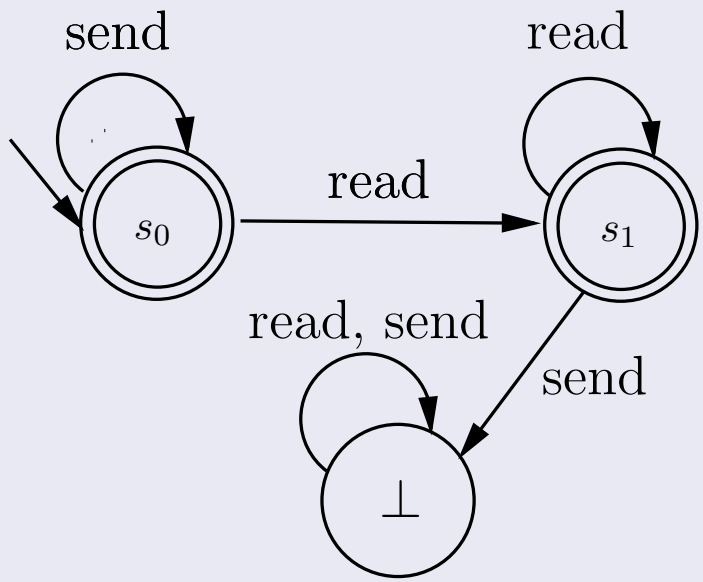


Figure: Automata for The Policy

The Policy Language ConSpec

- Adapted from PSLang of Erlingsson and Schneider
- Semantics given through security automata
- A restricted language to allow the formal treatment of several security related activities
  - Bounded domains for state variables to enable policy matching
  - Restricted language for state updates (e.g. no loops) for automatic proof generation

Policies

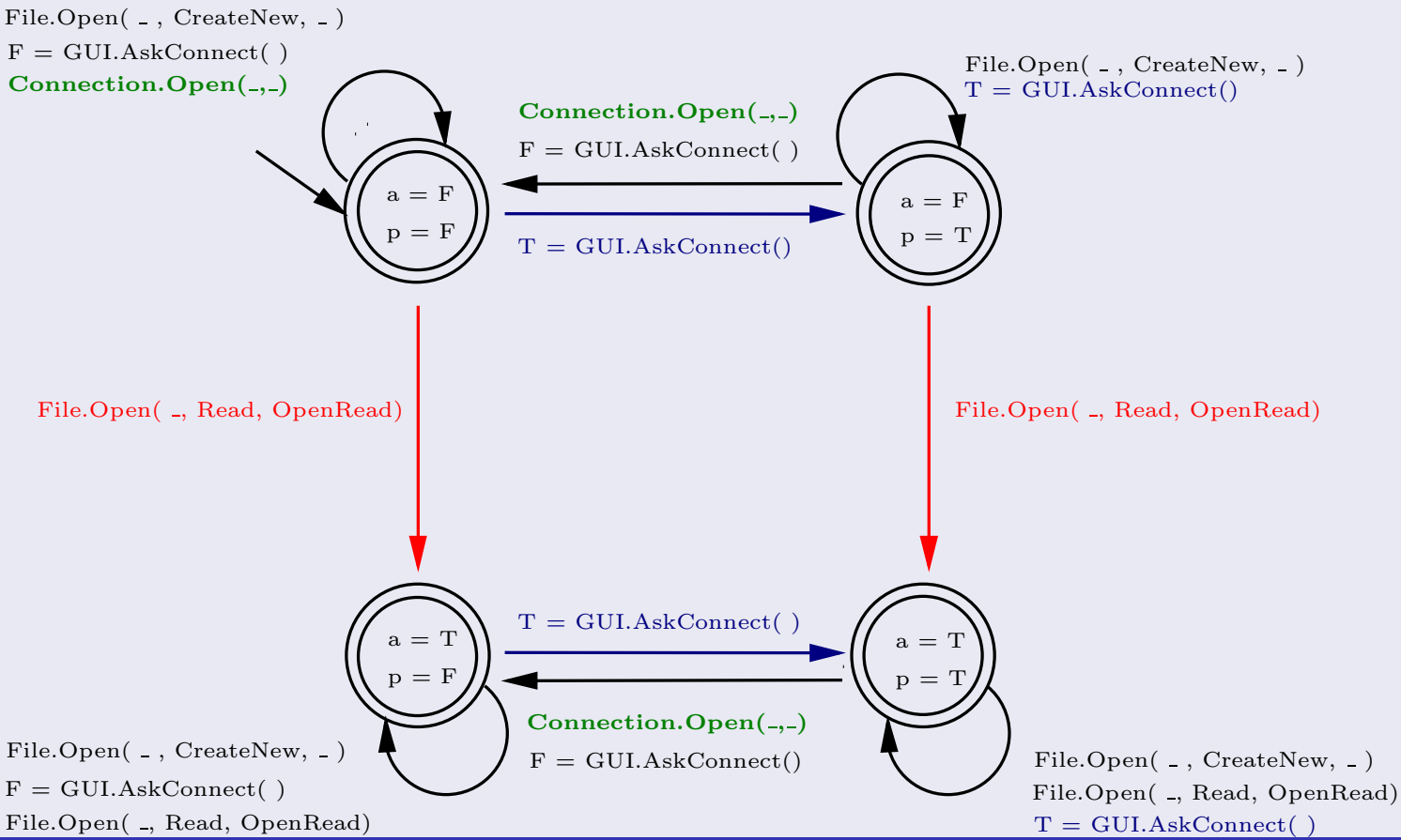
Example: ConSpec Language

"*Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened.*"

Policies

Example: ConSpec Language

"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."



Policies

Example: ConSpec Language

"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."

```
SECURITY   STATE
           bool accessed = false;
           bool permission = false;

BEFORE     File.Open(string path, string mode, string access)
PERFORM    mode.equals(CreateNew)        → { skip; }
           mode.equals(Open) &&
               access.equals(OpenRead) → { accessed = true; }

AFTER      bool answer = GUI.AskConnect()
PERFORM    answer → { permission = true; }
           !answer → { permission = false; }

BEFORE     Connection.Open(string type, string address)
PERFORM    !accessed || permission -> { permission = false; }
```

The Challenge

- Low level programs: We do not have access to source code at certification time and even possibly at time of proof generation
- Limited computational resources: Proof checking should occur on the mobile device
- "Push button" technique: Automatic proof generation is desirable for usability

Therefore, we want to simplify proof generation.
Why is proof generation and proof checking hard? Because we do not know why the program actually complies with the policy.

The Challenge

- Low level programs: We do not have access to source code at certification time and even possibly at time of proof generation
- Limited computational resources: Proof checking should occur on the mobile device
- "Push button" technique: Automatic proof generation is desirable for usability

Therefore, we want to simplify proof generation.
Why is proof generation and proof checking hard? Because we do not know why the program actually complies with the policy.

1. Take any program
2. Change it to adhere to the policy (even if it already does) Coming up: Monitor Inlining
3. Create proof (now we know approximately why the program complies with the policy)

The Challenge

- Low level programs: We do not have access to source code at certification time and even possibly at time of proof generation
- Limited computational resources: Proof checking should occur on the mobile device
- "Push button" technique: Automatic proof generation is desirable for usability

Therefore, we want to simplify proof generation.
Why is proof generation and proof checking hard? Because we do not know why the program actually complies with the policy.

1. Take any program
2. Change it to adhere to the policy (even if it already does) Coming up: Monitor Inlining
3. Create proof (now we know approximately why the program complies with the policy)

Runtime Monitoring

A *monitor* operates by observing the behavior of a target program and terminating the program when an action that violates the policy is about to occur.

Monitoring has been implemented in the following two ways:

- *external monitoring*: external entities that run in parallel with the target program (e.g. firewalls),
- *monitor inlining*: the program is *rewritten* to make it self-monitoring

Monitor Inlining

General purpose monitor inlining has been introduced by Evans and Twyman 1999/Erlingsson and Schneider 2000.

1. The monitor state is inserted into the program by the inliner.

   Embedded state: The concrete representation of the monitor state in the program, usually in the form of global program variables.

2. Code is inserted around relevant actions to check if the action violates the policy; program is terminated in case the action is violating and the embedded state is updated otherwise.

   Variants include:

   - Wrapping (Naccio'99)
   - Scattered (PSLang/POet'00)
   - Central (Polymer'05)

Policies

Example: ConSpec Language

"After applications access an existing file, they are required to obtain approval from the user each time a connection is to be opened."

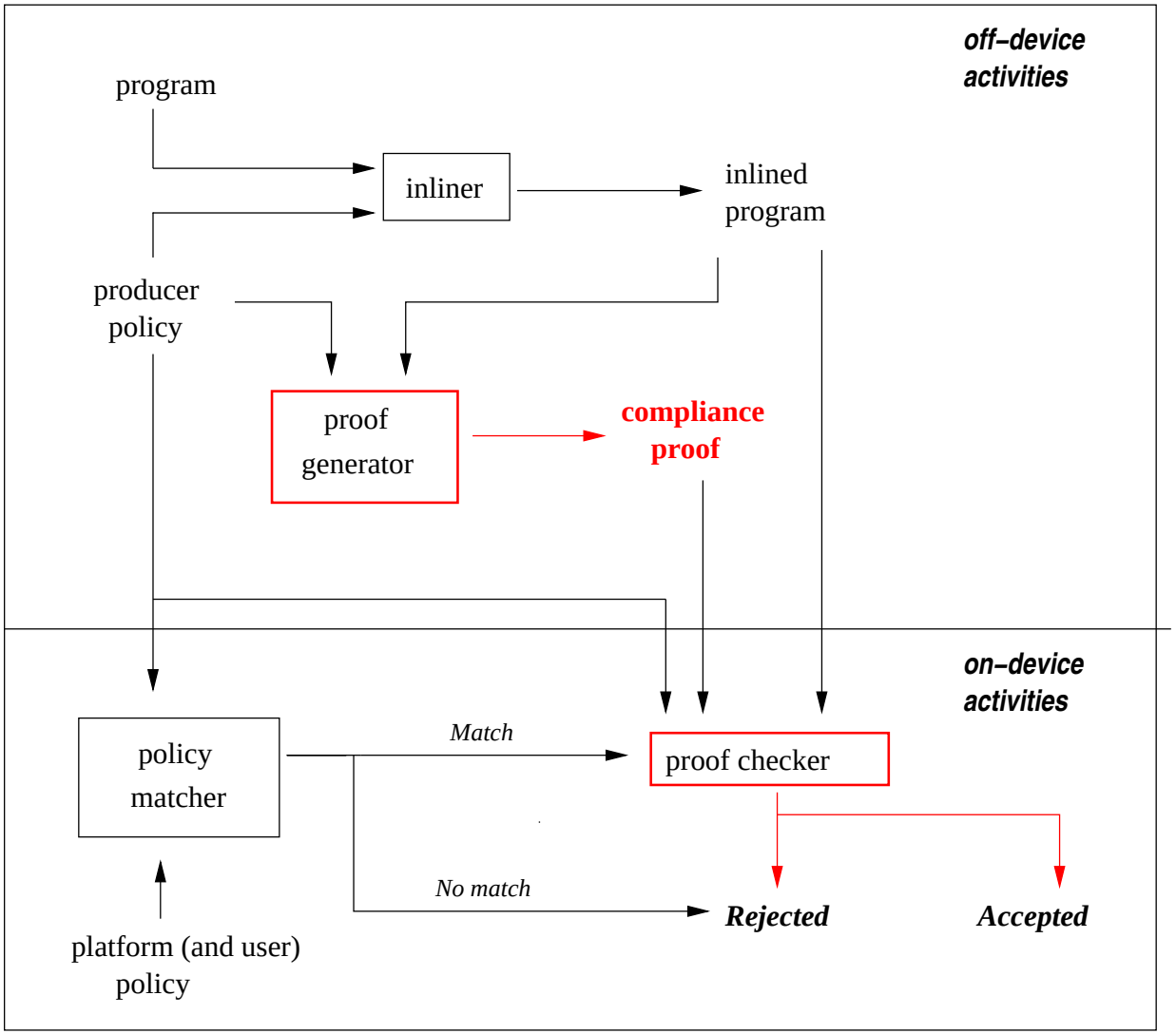
```
SECURITY   STATE
           bool accessed = false;
           bool permission = false;

BEFORE     File.Open(string path, string mode, string access)
PERFORM    mode.equals(CreateNew)       → { skip; }
           !mode.equals(CreateNew) → { accessed = true; }

AFTER      bool answer = GUI.AskConnect()
PERFORM    answer → { permission = true; }
           !answer → { permission = false; }

BEFORE     Connection.Open(string type, string address)
PERFORM    !accessed || permission -> { permission = false; }
```

Monitor Inlining

	Method before inlining	Method after inlining
L1	`aload r0`	`aload r0`
L2	`getfield gui`	`getfield gui`
L3	`invokevirtual GUI/AskConnect()Z`	`dup`
L4	`ireturn`	`astore r1`
L5		`invokevirtual GUI/AskConnect()Z`
L6		`istore r2`
L7		`aload r1`
L8		`instanceof GUI`
L9		`ifeq L12`
L10		`iload r2`
L11		`putstatic SecState/permission`
L12		`iload r2`
L13		`ireturn`

Figure: An application method inlined with the example policy

The Framework



Provably Correct Inlining

An inlined program is *correctly inlined* for policy $\mathcal{P}$, if the program adheres to the policy.

Provably Correct Inlining

An inlined program is *correctly inlined* for policy $\mathcal{P}$, if the program adheres to the policy.

Our mission

For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

Provably Correct Inlining

An inlined program is *correctly inlined* for policy $\mathcal{P}$, if the program adheres to the policy.

Our mission

For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

Our approach

Use Floyd-like logic to specify correct inlining!

Specifying Correct Monitor Inlining

The problem

How can we specify that a program has been correctly inlined for a given policy?

Specifying Correct Monitor Inlining

The problem

How can we specify that a program has been correctly inlined for a given policy?

Reformulation of the Problem

How can we specify that a program has an embedded monitor for the policy?

Specifying Correct Monitor Inlining

The target program is annotated in two steps:

Specifying Correct Monitor Inlining

The target program is annotated in two steps:

1. Level I: we insert a correct monitor into the program using specification variables.

Specifying Correct Monitor Inlining

The target program is annotated in two steps:

1. Level I: we insert a correct monitor into the program using specification variables.

2. Level II: we specify that an embedded state exists such that:
   - the embedded state is in agreement with the specified monitor immediately prior to execution of a security relevant action, and
   - the updates to the embedded state are made locally, that is by the method that executes the security relevant method call.

Level I Annotations

(Policy Annotations)

- Characterize policy adherence,

Level I Annotations
(Policy Annotations)

- Characterize policy adherence,
- Created using the policy:
  - The security state is represented by ghost variables (ghost state) and updated to mimic the automaton transitions,
  - The ghost state is asserted to be defined at critical points.

Level I Annotations

Policy

"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."

```
SECURITY   STATE
           bool accessed = false;
           bool permission = false;

BEFORE     File.Open(string path, string mode, string access)
PERFORM    mode.equals(CreateNew)        → { skip; }
           mode.equals(Open) &&
               access.equals(OpenRead)  → { accessed = true; }

AFTER      bool answer = GUI.AskConnect()
PERFORM    answer  → { permission = true; }
           !answer → { permission = false; }

BEFORE     Connection.Open(string type, string address)
PERFORM    !accessed || permission -> { permission = false; }
```

Level I Annotations

Target Program

L	$M[L]$
L1	`aload r0`
L2	`getfield gui`
L3	`dup`
L4	`astore r1`
L5	`invokevirtual GUI/AskConnect()Z`
L6	`istore r2`
L7	`aload r1`
L8	`instanceof GUI`
L9	`ifeq L12`
L10	`iload r2`
L11	`putstatic SecState/permission`
L12	`iload r2`
L13	`ireturn`

Figure: An target application method

Level I Annotations

Example: Level I Annotations

$A^{\mathrm{I}}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{Defined(gs)\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure: An application method with level I annotations for the example policy

Level I Annotations

Theorem: Correctness of Level I Annotations

Program T annotated with level I annotations for policy $\mathcal{P}$ is valid, if and only if T adheres to $\mathcal{P}$.

Level II Annotations

(Synchronisation Check Annotations)

- Characterize the existence of a concrete monitor in the program
- Obtained by extending level I annotations

Level II Annotations

(Synchronisation Check Annotations)

- Characterize the existence of a concrete monitor in the program
- Obtained by extending level I annotations

Synchronisation Assertion states the equality of the ghost state and the embedded state, and is asserted for every method at:

- method entry (pre-condition),
- method exit (post-condition),
- before each method call.

Level II Annotations

Example: Level I Annotations

$A^{\mathrm{I}}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{Defined(gs)\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Figure: An application method with level I annotations for the example policy

Level II Annotations

Example: Level II Annotations

$A^{II}[L]$	L	$M[L]$
$\{gs = \texttt{SecState}\}$	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\{Defined(gs) \wedge gs = \texttt{SecState}\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{gs := \delta_{\perp}(gs, a)\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
$\{gs = \texttt{SecState}\}$	L13	`ireturn`

Figure: An application method with level II annotations for the example policy

Level II Annotations

Theorem: Level II Characterization

The level II annotations of T for policy $\mathcal{P}$ with embedded state $\overrightarrow{ms}$ is valid if, and only if, $\overrightarrow{ms}$ is a method-local monitor for $\mathcal{P}$.

Correct Monitor Inlining

Given a program T, a policy $\mathcal{P}$ and an embedded state $\overrightarrow{ms}$,
how do we show that the level II annotated program is valid?

Correct Monitor Inlining

Given a program T, a policy $\mathcal{P}$ and an embedded state $\overrightarrow{ms}$,
how do we show that the level II annotated program is valid?

If T is a "nicely" inlined program then level II annotations can be completed to full annotations and the problem is reduced to checking local validity.

Checking local validity consists of

- constructing verification conditions using the axiomatic semantics of single instructions
- discharging the resulting verification conditions

Correct Monitor Inlining

Level III ("Full") Annotations for the Inliner

Full annotations generated by:

1. Adding the synchronization annotation as precondition to uninlined instructions
2. Propagating the synchronization annotation from the bottom to the top of the inlined blocks using a weakest precondition calculator

A program is "nicely" inlined if

- the problem of computing the weakest precondition of inlined blocks is decidable,
- the problem of discharging the verification conditions arising from the local validity of the full annotations is decidable

Proof of correct inlining can be constructed for nicely inlined programs.

Correctness of an Inliner

An inliner is a *well-behaved inliner* if it produces nicely inlined programs.

Correctness of a well-behaved inliner I can be proven by showing that given any program T and policy $\mathcal{P}$, a proof of inlining correctness can be constructed for the inlined program $\mathrm{I}(\mathrm{T}, \mathcal{P})$.

Conclusion

In this work we introduce a two-level annotation scheme on java bytecode programs where:

1. level I annotations characterize policy adherence,
2. level II annotations characterize existence of a (method-local) monitor in the program.

- The annotation scheme can be used to show that a program has been correctly inlined.

Conclusion

In this work we also describe how to compute full annotations for "nicely" inlined programs which reduce the problem of proving correct inlining to checking local validity of the fully annotated program.

- The annotation scheme can be used in a proof-carrying code setting for certifying monitor compliance to the code consumer.
- The annotation scheme can be used to show correctness of an inliner.

Future Work

- Constructing a proof-carrying code framework
- Extending the annotation scheme to handle multi-threaded programs
- Adding new features to the policy language, e.g. to define per-object policies
- Finding abstraction functions that identify the embedded state in unlined but yet self-monitoring programs
- Extending the correctness result to cover transparency

Open Problems

- Giving semantics to other policy scopes, e.g. to per-object policies
- Extending the annotation scheme to handle multi-threaded programs
- Finding abstraction functions that identify the embedded state in unlined but yet self-monitoring programs

The End

Thank You!